Dependency Issues in Aspect Weaving

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

Aditya Varma D

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

May, 2005

Certificate

This is to certify that the work contained in the thesis entitled " *Dependency Issues in Aspect Weaving* ", by *Aditya Varma D*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

May, 2005

(Dr. T.V.Prabhakar)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

Aspect-oriented technology is a new programming paradigm that is receiving considerable attention from research and practitioner communities alike. It deals with those concerns that crosscut the modularity of traditional programming mechanisms, and its objectives include a reduction in the amount of code written and higher cohesion. Aspect Oriented Programming is helpful in writing less tangled and easier to maintain code with a greater reuse. Weaving is one of the fundamental mechanisms of aspect-oriented systems. A weaver composes different aspects with the base system by determining and adapting all parts where aspect specific elements are needed eventually. Weaving of aspects could be done at compile time called static weaving or at runtime called dynamic weaving.

The drawback with currently available static and dynamic weavers is that there is no support for specifying the dependency of the aspects. Though some support is available in AspectJ, it is limited in its capabilities. When multiple aspects are woven, failure to preserve the dependencies amongst them might push the system into an inconsistent state. We analysed the problems that could arise when there are dependencies amongst the aspects being woven. We performed an analysis of whether "immediate or deferred weaving" should be done at runtime. Based on these, a dynamic weaver incorporating the dependency analysis has been built.

Acknowledgements

I take this opportunity to express my sincere gratitude to all those who contributed in making my thesis a success.

First of all I would like to thank the Almighty God for his support to me over these years. He is the person. whom I credit for the little I learned & achieved till now.

I would like to express my hearty gratitude towards my guide Dr.T.V.Prabhakar for his invaluable guidance. I consider him not just as a thesis supervisor, but as a "Guru" who has lended his advice and support in many aspects of my life. I thank him for accepting me to work under his supervision. His innovative ideas, constant encouragement and cool temperament motivated me to take my work to completion. His style of thinking and attitude inspired me a lot. In spite of his hectic schedule, he took his time off to attend to my problems and the papers which we had written. He is one of those few teachers whom I would never forget in my life.

I would like to thank the faculty members of the Computer Science Department for the knowledge they have imparted to me. I would like to give special thanks to my friends chaitanya, pratik, raghu, sudheer for helping me at times in my coding.

I would cherish the memories of the two years I had spent with my friends here. I could never forget the cricket matches, birthday bumps, treats and photo sessions. I thank my labmates especially Venkateswaran, who made my one year stay with them memorable.

I am forever obliged to my parents for their encouragement to me, and unwavering faith in my ability to succeed. Lastly, special thanks to my close friend Maturu Rama Krishna for his constant support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Aspect-oriented programming is a new evolution in the line of technology for seperation of crosscutting concerns. By "crosscutting concerns", we mean those program fragments that cannot be encapsulated in a single module and are scattered over various modules in the system. It is the technology that allows design and code to be structured to reflect the way developers want to think about the system. The goal of AOP is to make designs and code more modular, meaning that concerns are localized rather than scattered and have well-defined interfaces with the rest of the system. This provides us with the usual benefits of modularity, including efficient code, aspect reuse and a more easily maintainable code.

1.1 Overview

Object oriented programming has become mainstream over the last years, having almost completely replaced the procedural approach. One of the biggest advantages of object orientation is that a software system can be seen as being built of a collection of discrete classes. Each of these classes has a well defined task, its responsibilities are clearly defined. In an OO application, those classes collaborate to achieve the application's overall goal. However, there are parts of a system that cannot be viewed as being the responsibility of only one class, they cross-cut the complete system and affect parts of many classes. Examples might be locking in a distributed

application, exception handling, or logging method calls. Of course, the code that handles these parts can be added to each class separately, but that would violate the principle that each class has well-defined responsibilities. This is where AOP comes into play: AOP defines a new program construct, called an aspect, which is used to capture cross-cutting aspects of a software system in separate program entities. The application classes keep their well-defined responsibilities. Additionally, each aspect captures cross-cutting behavior.

AOP builds on existing technologies and provides additional mechanisms that make it possible to affect the implementation of systems in a crosscutting way. In AOP, a single aspect can contribute to the implementation of a number of procedures, modules or objects. The contribution can be homogenous, for example by providing a logging behavior that all the procedures in a certain interface should follow; or it can be heterogenous, for example by implementing the two sides of a protocol between two different classes.

Initially, weaving of aspects started with static weaving - weaving done at compile time. AspectJ [8], Hyper/J [14] are examples of static weaving of aspects. AspectJ is an extended language of Java facilitating features and constructs for writing and inserting aspects at compile time. An aspect is defined in terms of "Join points", "Point cuts" and "advices". A join point represents a well-defined point in the execution of a program. Point cut represents a collection of join points. An advice represents the behavior to be executed at the join point. Weaving of an aspect into base code requires access to the source code. The base code is transformed by embedding calls to the advice methods. For every dynamic join point in the aspect there is a corresponding static shadow in the source code. Each piece of advice is matched with each static window. If there is a match, a call to the advice method is embedded into the source file.

The main drawback of static aspect weavers is that they do not facilitate the insertion or deletion of aspects during the execution of system. The decision of which aspect to be weaved has to be done beforehand, leaving no chance to reconfigure the

system at runtime without aborting and restarting it. Once an aspect is statically woven, identifying aspect specific statements which are added during weaving is difficult. This led to the advent of dynamic weaving of aspects. The two most widely used approaches for dynamic weaving are: "byte code manipulation" and the "interpreter approach". There is also another approach in use called "proxy approach" though not widely used as compared to the former ones. Currently there are tools basing on the above-mentioned approaches like PROSE [12] , JAC [11] , Wool [18] , AspectWerkz [2] , CLAW [4], AOP/ST, Jasco etc.

1.2 Motivation and Goals

Aspects not only interact with the primary abstraction but also with other aspects. Depending on their interactions, aspect dependencies can be classified as "orthogonal", "uni-directional", "cyclic-dependent". Most of the weaving tools do not consider inter-dependencies that could exist among aspects. They just follow a fixed order (for example, PROSE executes aspects in the decreasing order of their insertion time). This could result in a violation of dependencies leading to inconsistencies in the system. Though some support is available in AspectJ, it is limited in its capabilities. Since weave order cannot be specified, weaving multiple aspects at a time or even a single aspect, deviating from the dependency order would lead to problems in the system. Failure to preserve dependency order of aspects would lead to inconsistencies in various states of the system. These undesirable behavior is termed "emergent properties".

Our goal was to analyze the dependencies that could exist among the aspects and develop a weaver with the feature of preserving aspect dependencies. The dependencies were interpreted from a database transaction point of view. We analysed the consistency problems that can result from the aspect dependencies. Using these analyses, we decide, given a new aspect that needs to be added/withdrawn to/from the system, whether it can be woven/removed immediately or whether it needs to wait for the currently executing methods or advices to return. Our implementation is based on the tool "PROSE". The basic mechanism for representing pointcuts,

advices have been incorporated in the similar lines as PROSE. In case the weaver determines that, an aspect can be weaved due to the dependencies, a feedback is provided to the aspect developer specifying the aspects which play part in the violation of dependencies.

1.3 Organisation of the Thesis

The thesis is organised as follows

- In chapter 2 we describe the drawbacks of the existing object-oriented paradigm and the need for Aspect Oriented programming. We briefly discuss the features and working of AOP.
- In chapter 3 we discuss the terminology of aspects & weaving and the mechanism of static weaving. AspectJ is used as the base for discussing the basic terminology.
- In chapter 4 we describe the disadvantages of static weaving and the advent of dynamic weaving. We also describe the mechanisms used in dynamic weaving.
- In chapter 5 we present the dependencies and the problems that arise in case of static weaving. We illustrate an analysis, to be done before the weaving of an aspect. We do the same for dynamic weaving. We end the chapter with a brief note on the dependency issues during the withdrawal of aspects.
- In chapter 6 we describe our implementation, its architecture and working.
- In chapter 7 we present the conclusions along with the possible extensions.

Chapter 2

Introduction to AOP

Design of a software system involves addressing of the primary functionality. For instance, if we consider a banking application, the primary requirement is to manage the banking transactions made by the customer, manage the customer accounts etc. In a supply chain management system, the core functionality is to manage the inventory, obtain the product availability on an order etc. Irrespective of the core primary functionality, the system has to address some features such as logging, authorization, persistence which are required by many of the core modules.

A *concern* is a particular goal. concept or area of interest that must be addressed in order to satisfy the system goals. *Concerns* are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. A software system is viewed as a set of concerns. They can be functional, like features or business rules, or nonfunctional, such as synchronization and transaction management. For instance, a banking application is a realization of the following concerns: customer account management, interest computation, inter-banking transactions, persistence of the data, authorization of transactions etc. In addition to primary concerns, a software project needs to address process concerns, such as comprehensibility, maintainability, traceability, and ease of evolution.

Concerns are classified into two categories: core concerns represent the heart of the system. They are concerned with the central functionality of a module.

Crosscutting concerns represent system-level requirements that "cuts across" many different classes and methods. An application may need to address crosscutting concerns, such as logging, profiling, tracing, persistence, security management to name a few. All of these concerns crosscut several modules. For example, in a banking system, authentication concern needs to be addressed in every module involved in a transactional access; persistence concern affects every stateful object. Figure 2.1 represents a system as a composition of multiple concerns.

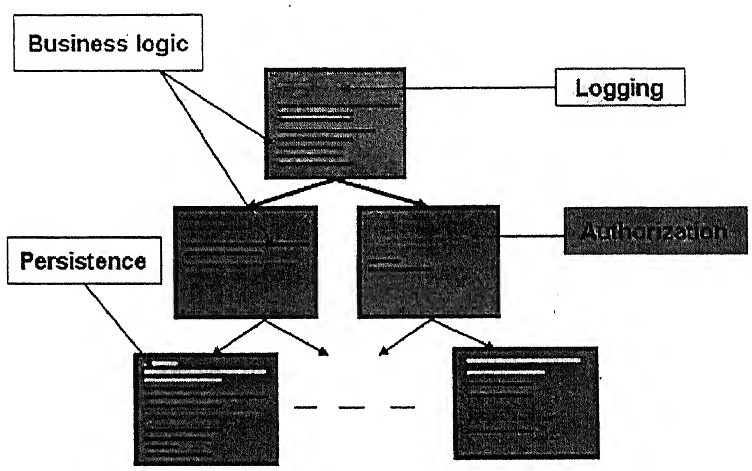


Figure 2.1: *View of a system as a composition of multiple concerns.*

From the above figure, we can see how the implementation modules in a system each address both system-level and business concerns. Each module addresses some element from each of the concerns. We can clearly see that the concerns got tangled together. The orthogonality of the concerns is lost.

2.1 Identifying system concerns

Design and implementation of a system begins with identification of concerns. Once the core and crosscutting concerns are identified, each of them can handled independently thereby reducing the complexity of design. So, initially the set of requirements have to be decomposed into concerns. Figure 2.2 uses the analogy of a light beam

passing through a prism to illustrate the process of decomposing the requirements into a set of concerns. We pass a light beam of requirements through a concern identifier prism and the concerns get separated out.

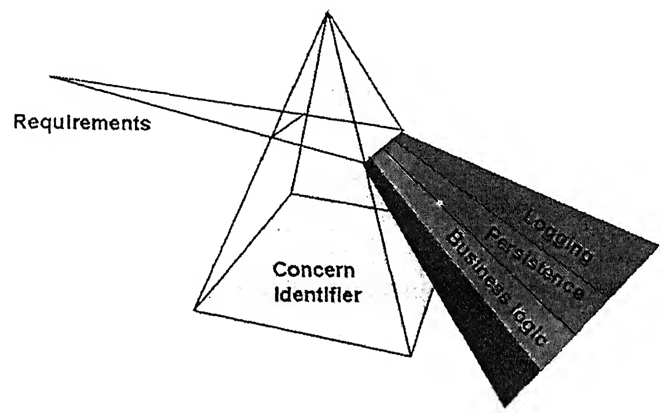


Figure 2.2: *Light-beam analogy for concern decomposition*

At the requirements phase, each of the requirements appear to be a single unit. By applying the concern identification process, we can separate out the individual core and crosscutting concerns required to fulfill the requirements.

A beautiful way to view the concern decomposition is to project the concerns onto a N-dimensional concern space. In such a space, each concern would lie along a dimension. Figure 2.3 shows a three-dimensional concern space with the business logic core concern and the persistence and logging crosscutting concerns as the dimensions.

Realizing the concern decomposition in this view clearly gives us the understanding that, at design phase all the concerns are mutually independent and each can evolve separately with no relation to the others. For example, changing the persistence requirement from a relational database to an object database should not affect the business logic or security requirements.

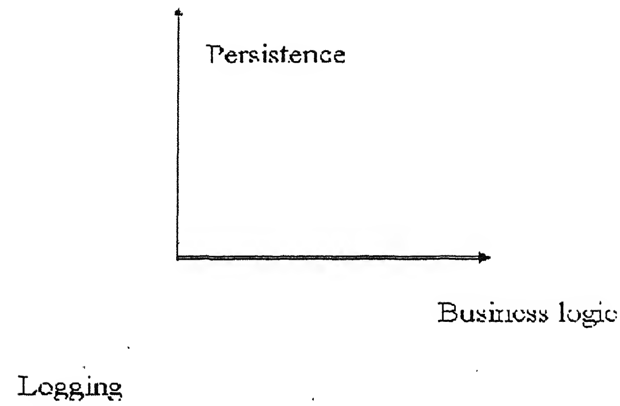


Figure 2.3: *Concern decomposition: a multidimensional view*

Identification of concerns is a vital part of the design phase in the development of a software system. Once the concerns are identified, each of them can be addressed independently. As we see further, the current programming paradigms fail to preserve the orthogonality of the concerns in the implementation phase.

2.2 A two-dimensional solution

As said before, crosscutting concerns "cut across" many modules. The problem is that, current implementation techniques tangle and scatter crosscutting concerns into the individual core modules. This fact is illustrated by the figure 2.4.

The figure depicts a 2-dimensional object model. In this implementation space, concerns are mapped onto the core modules(objects). But once imagine what happens when you try to render a 3 dimensional space in a 2 dimensional table. You get repetition. The same is the case with cross-cutting concerns in code. They are scattered because the 2-dimensional object model is short one dimension. The missing dimension is where such concerns should materialize. While we may naturally separate the individual requirements into mutually independent concerns during the design phase, current programming methodologies do not allow us to retain the separation in the implementation phase.

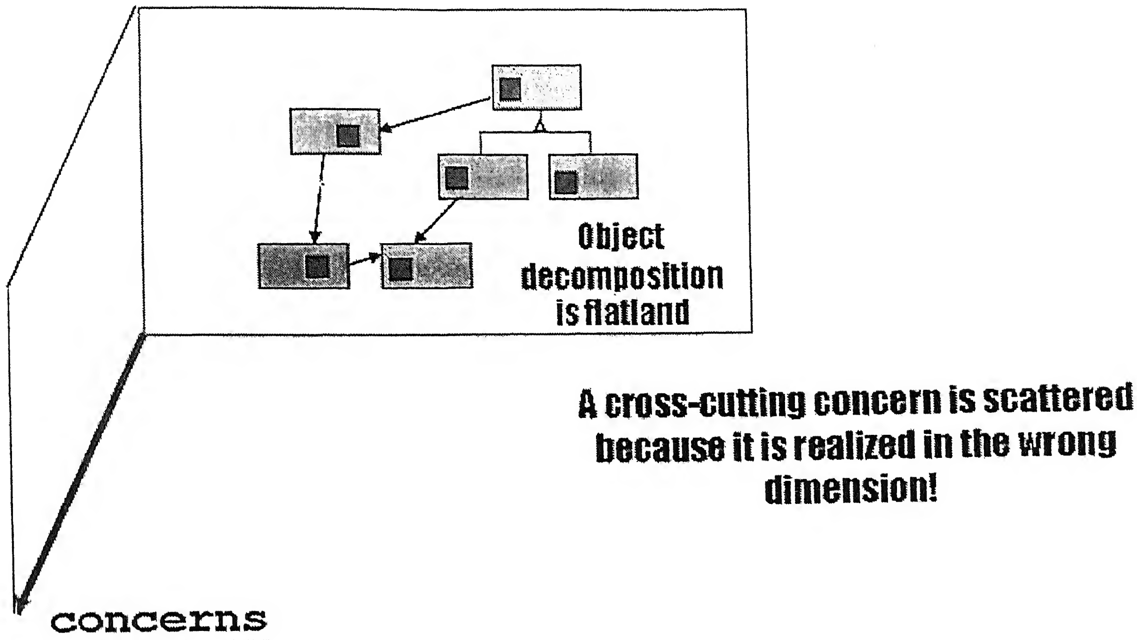


Figure 2.4: *2-dimensional object model representation of implementation space.*

2.3 Where does the problem lie?

The object oriented paradigm (OOP) has been heralded as the savior to the problems faced in software development. It has undoubtedly raised our ability to design and maintain large complex software systems. The efficiency of OOP is a result of the high level of modularization that can be achieved by using separate objects with their own state, behavior, and responsibility structures. This modularization makes possible the creation of extensive libraries and a high level of code reuse. As it stands, OOP is efficient in modularizing core concerns only. When it tends to modularize the crosscutting concerns, such modules are closely coupled with the core modules thereby leading to more "unwanted side effects".

In OOP, the core modules can be loosely coupled through interfaces, but the same approach cannot be followed for crosscutting concerns. The reason is that every concern has two parts: the server-side and the client-side. The term server

and client represent the objects providing a certain set of services and the objects using those services respectively.. OOP modularizes the server part quite well in classes and interfaces. However, when the concern is of a crosscutting nature, the client part, consisting of the requests to the server, is spread over all of the clients.

As an example, let's look at a typical implementation of a crosscutting concern in OOP: a display-tracking module that provides its services through an abstract interface. The use of an interface loosens the coupling between the clients and the implementations of the interface. Clients using the display-tracking services through the interface are for the most part oblivious to the exact implementation they are using. Any changes to the implementation will not affect the clients themselves. Similarly, replacing one display-tracking implementation with another is just a matter of instantiating the right kind of implementation. The result is that one implementation can be switched with another with little or no change to the individual client modules. This configuration, however, still requires that each client have the embedded code to call the API. Such calls will need to be in all the modules requiring the display-tracking service and will be mixed in with their core logic. The overall effect is an undesired tangling between all the modules needing the service and the display-tracking module itself. The problem gets more aggravated when many such crosscutting concerns are handled in a core module. Figure 2.5 in the next page clearly illustrates the above.

2.4 Implementing crosscutting concerns in non-modularized systems

Let's now take a more detailed look at the nature of crosscutting concerns. The implementation of crosscutting concerns often becomes complicated by tangling it with the implementation of core concerns and scattering a crosscutting concern in multiple core modules. Listing 1.1, shows the implementation of a business logic module using the object oriented methodology.

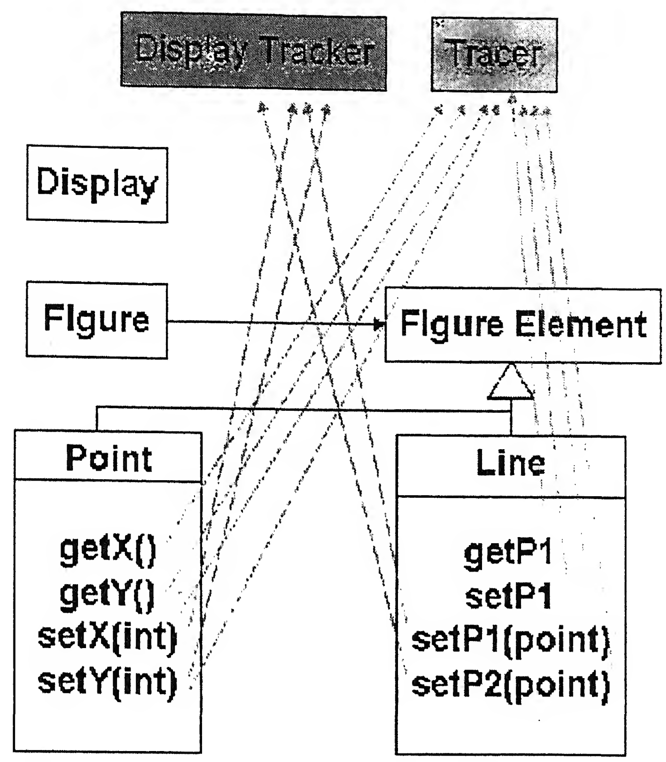


Figure 2.5: *Implementation of tracing and display-tracking concerns using conventional techniques*

Listing 2.1: Business logic implementation

```
public class Businesslogic {
... Core data members
... Log stream
... Cache update status
public void businessOperation(param1,param2,...)
{
... Enforce authorization
... Check integrity constraints
... Cache management
... Logging operation-initialization
```

```
... Core functionality
... Logging operation-exit
}
}
```

Form the above skeleton implementation, we can see that the core business module has to handle concerns such as logging, cache management, integrity checking, authorization other than the core concern. These other concerns are the cross-cutting concerns which would be addressed by many such core business modules. As said before, though concerns are regarded as independent in design phase, the orthogonality is not preserved at implementation phase.

2.4.1 Symptoms of non-modularization

The symptoms of non-modularization can be of two types: code tangling and code scattering.

Code tangling

Code tangling is caused when multiple concerns are interleaved in a single module i.e., presence of elements form multiple concerns' implementation in a single core module. In the listing 2.1, concerns such as logging, authorization are addressed in the business module.

Code scattering

Code scattering is caused when a single concern affects multiple modules. It is in turn classified into two types: duplicated code blocks and complementary code blocks.

Duplicated code blocks occur when identical code related to a concern is placed in multiple modules. For example, in an application where all the business transactions to be logged, logging functionality has to be placed in many modules involved in accessing business data.

Complementary code blocks occur when complementary parts of a concern are implemented in several modules. For example, in an application where the users need

to be checked for access privileges before permitting to access a resource - authentication is handled by one module, authorization by another and so on.

2.4.2 Cost of crosscutting concerns

The implications of code tangling and code scattering on software design and development are:

- Reduced understandability - code scattering leads to redundant code in many places. There would be no explicit structure to map a concern to its implementation. Tracing a concern to its implementation requires examining of all the modules.

- Decreased adaptability - modifying a concern or adding a new one would require modifying many subsystems. This modification should be done consistently, lest there would be some modules unmodified. Adding of a new concern is difficult as it would require reconstruction of many modules.

- Decreased reusability - Component code is tangled with specific crosscutting code in many modules. If a system wants to reuse the core functionality of such module, it can't do so as reusing such module would require using the crosscutting functionality as well. But the system may not require those peripheral concerns. The result is the loss in the reusability provided by the OOP paradigm.

- Lower productivity - Simultaneous implementation of multiple concerns shifts the focus from the implementation of main concern to the peripheral concerns. The lack of focus then leads to lower productivity as developers are sidetracked from their primary objective in order to handle the crosscutting concerns. Further, since different concern implementations may need different skill sets, either several people will have to collaborate on the implementation of a module or the developer implementing the module will need knowledge of each domain.

2.5 The architect's dilemma

Good system architecture considers present and future requirements to avoid a patchy-looking implementation. Therein lies a problem, however. Predicting the future is a difficult task. If you miss future crosscutting requirements, you'll need to change, or possibly re-implement, many parts of the system. On the other hand, focusing too much on low-probability requirements can lead to an over-designed, confusing, bloated system. Thus a dilemma for system architects: How much design is too much? Should I lean towards under-design or over-design?

The architect seldom knows every possible concern the system may need to address. Even for requirements known beforehand, the specifics needed to create an implementation may not be fully available. The architect thus faces the under/over-design dilemma.

2.6 Advent of AOP

Several methodologies have been developed to modularize crosscutting concerns like subject-oriented programming [16], Aspect-oriented programming [9], meta-object protocols [3], Composition filters [5]etc. Aspect-oriented programming is the most developed among these. Aspect-oriented programming (AOP) grew out of a recognition that typical programs often exhibit behavior that does not fit naturally into a single program module, or even several closely related program modules. AOP at its core, is a programming technique that lets the programmers implement individual concerns in a loosely coupled fashion, and combine these implementations to form the final system. Aspect-Oriented Programming was named by Gregor Kiczales and his group working at Palo Alto Research center.

AOP creates systems using loosely coupled, modularized implementations of crosscutting concerns(behavior that cuts across the typical divisions of responsibility). OOP, in contrast, creates systems using loosely coupled, modularized implementations of common concerns. AOP complements object-oriented programming

by facilitating another type of modularity that pulls together the widespread implementation of a crosscutting concern into a single unit. These units are termed aspects, hence the name aspect-oriented programming. By compartmentalizing aspect code, crosscutting concerns become easy to deal with. Aspects of a system can be changed, inserted or removed at compile time, and even reused.

The definition of "aspect" has evolved over time. The current working definition is (May 99, Gregor Kiczales):

An aspect is a modular unit that cross-cuts the structure of other modular units. Aspects exist in both design and implementation. A design aspect is a modular unit of the design that cross-cuts the structure of other parts of the design. A program or code aspect is a modular unit of the program that cross-cuts other modular units of the program.

AOP differs most from OOP in the way it addresses crosscutting concerns. With AOP, each concern's implementation remains unaware that other concerns are "aspecting" it. For example, a credit card processing application doesn't know that the other concerns are logging or authenticating its operations. That represents a powerful paradigm shift from OOP. As a result, the implementation of each individual concern evolves independently. AOP attempts to realize cross-cutting concerns as first-class elements and eject them from the object structure into a new dimension of software development. This can be illustrated by Figure 2.6

Various tools have been developed basing on AOP methodology. AspectJ [17] was the first tool developed by the group at PARC. It is the most mature and fully featured framework available today. AspectJ is an aspect-oriented extension to the Java programming language. Some of AspectJ's authors mention about performance optimizations that drove a 768-line program into 35,213 lines. Rewritten with aspect-oriented techniques, the code shrank back to 1,039 lines while retaining most of the performance benefits. Later, many tools have been developed for static and run-time weaving such as Hyper/J [14], AspectWerkz [2], JAC [11], PROSE [1] etc.

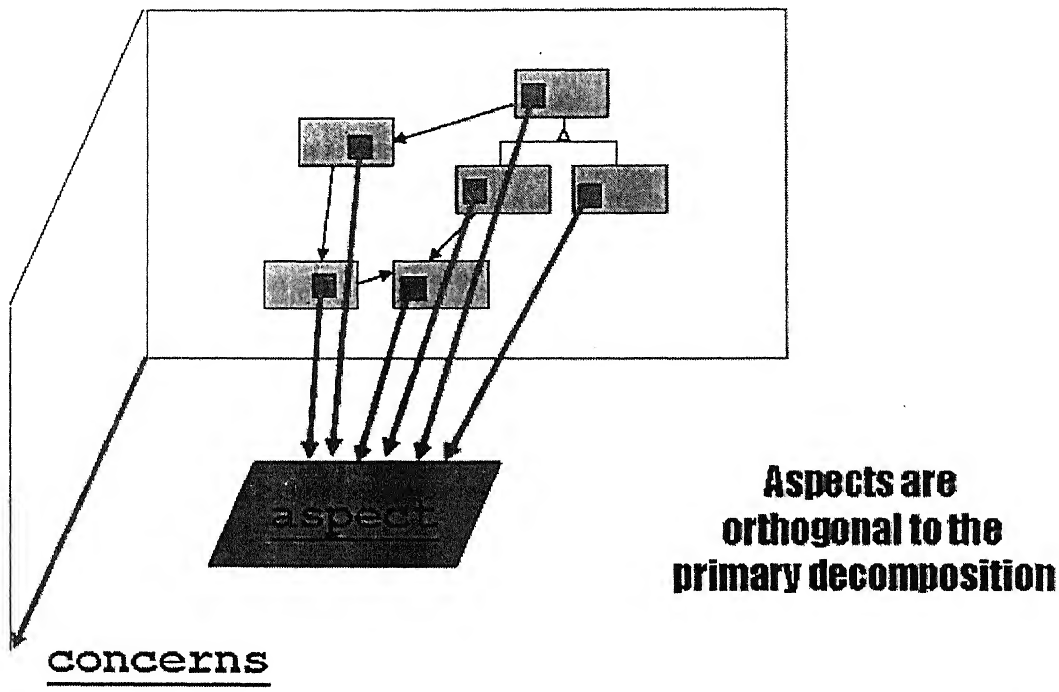


Figure 2.6: *AOP: Opening a new dimension*

2.6.1 AOP mechanism

AOP involves three distinct development steps:

1. Aspectual decomposition: Decompose the requirements to identify crosscutting and common concerns. Separate module-level concerns from crosscutting system-level concerns. For example, in a credit card processing application, we identify three concerns: core credit card processing, logging, and authentication.

2. Concern implementation: Implement each concern separately. In case of a credit card processing application, we implement the core credit card processing unit, logging unit, and authentication unit.

3. Aspectual recomposition: In this step, an aspect integrator specifies recomposition rules by creating modularization units – aspects. The recomposition process, also known as weaving or integrating, uses this information to compose the final system. In the case of a credit card processing application, we specify, in a language

provided by the AOP implementation, that each operation's start and completion be logged. We also specify that each operation must be authenticated before it proceeds with the business logic.

Just like any other programming methodology implementation, an AOP implementation consists of two parts: a language specification and an implementation. The language specification describes language constructs and syntax. The language implementation verifies the code's correctness according to the language specification and converts it into a form that the target machine can execute.

AOP language specification

At a higher level, an AOP language specifies two components:

- Implementation of concerns: Mapping an individual requirement into code so that a compiler can translate it into executable code. Since implementation of concerns takes the form of specifying procedures, we can use traditional languages like C, C++, or Java with AOP.

- Weaving rules specification: How to compose independently implemented concerns to form the final system. For this purpose, an implementation needs to use or create a language for specifying rules for composing different implementation pieces to form the final system. The language for specifying weaving rules could be an extension of the implementation language, or something entirely different.

AOP language implementation

AOP language compilers perform two logical steps:

1. Combine the individual concerns.
2. Convert the resulting information into executable code

The process that combines the individual concerns according to the weaving rules is called "weaving" and the processor doing this job is called a "weaver". The weaving rules are defined in aspects that are separate entities from the individual

core modules. This separation makes it possible to change the woven system simply by providing alternative weaving rules in the aspects. The working of a weaver is dealt in the succeeding chapters.

2.6.2 An Example

To get a better feeling of aspect-oriented programming, let's look at a small code sample in listing 2.2

Listing 2.2: Log calls manually inserted into every method

```
public void doGet(JspImplicitObjects theObjects) throws ServletException
{
  logger.entry("doGet(...)");
  JspTestController controller = new JspTestController();
  controller.handleRequest(theObjects);

  logger.exit("doGet");
}

public class logger{

public void entry(String str)
{System.out.println("method entered"+str);}

public void exit(String str)
{System.out.println("method exited"+str);}

}
```

In the above listing, the method entry/exit are traced by invoking methods of logger object. If several methods(say, all methods in a class "cls") have to be traced, code scattering will occur due to writing of redundant code. If the same is to be done using AOP, it would look like listing 2.3. The language used is AspectJ. The

syntax many not be understandable. The example is meant to give a flavor of what AOP looks like. The calls are replaced by a single aspect that automatically logs both parameters values along with method entries and exits.

Listing 2.3: Log calls automatically applied to every method

```
public aspect AutoLog{

pointcut loggableCalls() : execution(* cls.*(..));

before() : loggableCalls(){
  System.out.println("method entered"+
      thisJoinPoint.getSignature().toString());}

after() : loggableCalls(){
  System.out.println("method exited"+
      thisJoinPoint.getSignature().toString());}
}
```

Now, we have preserved the independence of concerns at implementation phase. Each of the concerns can evolve separately without affecting the core concerns. Code scattering doesn't occur as the specification of weaving resides only in the aspect containing the weaving rules. A point to be noticed here is that while using AspectJ, we need to explicitly specify the source files (both aspect and class) that we want to include in a given compilation. By specifying the files that are included in a given compilation, you can also plug and unplug various aspects of your system at compile time. For instance, by including or excluding the logging aspect described earlier from a compilation, the application builder can add or remove method tracing.

2.6.3 Advantages of AOP

Grady Booch describes aspect-oriented programming as one of three movements that collectively mark the beginning of a fundamental shift in the way software is designed and written. AOP addresses a problem space that object-oriented and

other procedural languages have never been able to deal with. Of course, AspectJ does have a learning curve. As with any language or language extension, it has subtleties that you need to grasp before you can leverage its full power. However, the learning curve is not too steep. The advantages of AOP are:

- Modularized implementation: The effort that goes into defining all of an application's requirements for a service like logging means the services are better-defined, better-coded and emerge offering richer functionality. It also addresses problems created when code is written to perform more than one function, as changes to one function can impact the other, a problem AOP dubs "tangling". Every concern is addressed separately with loose coupling and high cohesion.

- More Productivity: Making the effort to create discrete aspects allows a development team to assign an expert to the job, so that the best people for the job can exercise their skill and experience.

- Efficient code reuse and maintenance: In a typical object-oriented project different coders will often start with the same code to create a service like logging, then each diverge in their individual implementations of the logging services required to support an individual object. By creating a single piece of code, AOP offers a simple route to consistency, which enhances future re-usability and eases maintenance. Instead of re-writing logging routines throughout an application AOP makes it possible to re-write just the logging aspect and have the new functionality serve the entire application. Previously, without AOP, code reusability was less as reusing a core module would require using the implementations of the crosscutting concerns addressed by it. With AOP, all the aspects are loosely coupled with the core modules resulting in efficient code reuse.

- Late binding of design decisions: AOP solves the architect's under/over design dilemma. It makes applications architecturally flexible easily allowing for changes late in the development lifecycle. Any new requirements to be added can be written as aspects and embedded into the system.

Chapter 3

Aspect terminology & Weaving mechanisms

Before going into the mechanisms of aspect weaving, we would have a quick glance at the aspect terminology. AspectJ [8] was the foremost implementation of Aspect oriented programming. Since then, the various tools being developed and have been developed, followed the semantics and terminology used in AspectJ. So, our discussion would base on AspectJ to explain the aspect semantics.

AspectJ is a simple and practical extension to the Java programming language that builds upon the object model of Java with enhancements that allow aspect-oriented programming techniques to be used. It is compiled into standard Java byte code, and it is able to run on any Java platform.

AspectJ adds to Java just one new concept, a join point – and that's really just a name for an existing Java concept. It adds to Java only a few new constructs: pointcuts, advice, inter-type declarations and aspects. Pointcuts and advice dynamically affect program flow, inter-type declarations statically affects a program's class hierarchy, and aspects encapsulate these new constructs. We would be looking at join-points, pointcuts, advices excluding inter-type declarations. Inter-type declarations were not used in the later tools that have been developed.

3.1 AspectJ semantics

AO languages have three critical elements: a join point model, a means of identifying join points, and a means of affecting implementation at join points.

Join Point

Join points are certain well-defined points in the execution flow of the program. AspectJ has several kinds of join points like method call join point, method execution join point, field access join point, field modification join point etc. For instance, A method call join point is the point in the flow when a method is called, and when that method call returns. Each method call itself is one join point. The lifetime of the join point is the entire time from when the call begins to when it returns (normally or abruptly). The following code shows an example of the method execution join point :

```
public class Test {
        ...
    static void foo()
    {
        System.out.println("hi");
    }
}
```

The join point for the execution of the method foo is the whole method body.

Pointcut

Pointcuts capture, or identify, join points in the program flow. In addition to matching join points, certain pointcuts can expose the context at the matched join point. For example, the pointcut

```
call(void Point.setX(int))
```

picks out each join point that is a call to a method that has the signature void Point.setX(int) - that is, Point's void setX method with a single integer parameter.

Pointcut designators identify particular join points by filtering out a subset of all the join points in the program flow. For example:

```
call(void Point.setX(int)) ||
call(void Point.setY(int))
```

picks out each join point that is either a call to setX or a call to setY.

Pointcuts can identify join points from many different types - in other words, they can crosscut types. For example,

```
call(void Point.setX(int))  ||
call(void Point.setY(int))  ||
call(void Line.setP1(Point))  ||
call(void Line.setP2(Point));
```

picks out each join point that is a call to one of five methods (the first of which is an interface method, by the way).

AspectJ allows programmers to define their own named pointcuts with the pointcut form. So the following declares a new, named pointcut:

```
pointcut move():
    call(void Point.setX(int))  ||
    call(void Point.setY(int))  ||
    call(void Line.setP1(Point))  ||
    call(void Line.setP2(Point));
```

and whenever this definition is visible, the programmer can simply use move() to capture this complicated pointcut.

The previous pointcuts are all based on explicit enumeration of a set of method signatures. We sometimes call this *name-based crosscutting.* AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. We call this *property-based crosscutting.* The simplest of these involve using wildcards in certain fields of the method signature. For example, the pointcut

```
call(void Figure.make*(..))
```

picks out each join point that's a call to a void method defined on Figure whose the name begins with "make" regardless of the method's parameters. The pointcut

```
call(public * Point.* (..))
```

picks out each call to Point's public methods.

But wildcards aren't the only properties AspectJ supports. Another pointcut, cflow, identifies join points based on whether they occur in the dynamic context of other join points. So

```
cflow(move())
```

picks out each join point that occurs in the dynamic context of the join points picked out by move(), our named pointcut defined above. So this picks out each join point that occurs between, when a move method is called and when it returns (either normally or by throwing an exception).

Advice

Advice declarations are used to define additional code that runs at join points. AspectJ has several different kinds of advice. Before advice runs as a join point is reached, before the program proceeds with the join point. For example, before advice on a method call join point runs before the actual method starts running, just after the arguments to the method call are evaluated.

```
before(): move() {
    System.out.println("before advice");
}
```

After advice on a particular join point runs after the program proceeds with that join point. For example, after advice on a method call join point runs after the method body has run, just before control is returned to the caller. Because Java programs can leave a join point 'normally' or by throwing an exception, there are

three kinds of after advice: after returning, after throwing, and plain after (which runs after returning or throwing, like Java's finally).

```
after() returning: move() {
    System.out.println("after advice");
}
```

Around advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point.

Exposing Context in Pointcuts

Pointcuts not only pick out join points, they can also expose part of the execution context at their join points. Values exposed by a pointcut can be used in the body of advice declarations.

An advice declaration has a parameter list (like a method) that gives names to all the pieces of context that it uses. For example, the after advice

```
after(Point p, int x, int y) returning:
        ...SomePointcut... {
    ...SomeBody...
}
```

uses three pieces of exposed context, a Point named p, and two integers named x and y.

The body of the advice uses the names just like method parameters, so

```
after(Point p, int x, int y) returning:
        ...SomePointcut... {
    System.out.println(p + "coordinates:" + x + ":" + y);
}
```

The advice's pointcut publishes the values for the advice's arguments. The three primitive pointcuts this, target and args are used to publish these values. So now we can write the complete piece of advice:

```
after(Point p, int x, int y) returning:
        call(void Point.setXY(int, int))
        && target(p)
        && args(x, y) {
    System.out.println(p + "coordinates:" + x + ":" + y);
}
```

The pointcut exposes three values from calls to setXY: the target Point - which it publishes as p, so it becomes the first argument to the after advice – and the two integer arguments – which it publishes as x and y, so they become the second and third argument to the after advice.

So the advice prints the figure element that was moved and its new x and y coordinates after each setXY method call.

Aspect

Aspects wrap up join points, pointcuts and advices in a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, and initializers in addition to the crosscutting members. Aspects can declare themselves to be abstract. Aspects can extend classes and abstract aspects, as well as implement interfaces. Aspects can have access specifications.

An example of an aspect in AspectJ:

```
aspect FaultHandler {

    private void reportFault()
    { System.out.println("Failure"); }

    pointcut services(Server s): target(s) && call(public * *(..));
```

```
    before(Server s): services(s)
    { if (s.disabled) throw new DisabledException(); }

    after(Server s) throwing (FaultException e): services(s)
    {
     s.disabled = true;
     reportFault();
    }
}
```

Whenever a crosscutting behavior is to be embedded into the system, initially the join points at which we want to extend or modify the behavior should be identified. To implement the design of the required behavior, an aspect that serves as a module to contain the overall implementation needs to be written. Then, within the aspect, pointcuts to capture the desired join points are written. Finally, advices are written for each pointcut specifying the action that needs to happen upon reaching the join points.

When a join point matches the pattern specified by an aspect's pointcut, advice is applied through a composition process called weaving to the matching join point in the method body.

3.2 Mechanisms of weaving

There are three types of aspect weaving :

- Static or compile-time weaving
- Load time weaving
- Dynamic or run-time weaving

Initially, weaving of aspects started with static weaving. The initial aspect weavers developed basing on the separation of concerns principle like AspectJ, Hyper/J, AspectC are all static weavers. At this moment we would be discussing only static and load-time weaving. Dynamic weaving is dealt in the next chapter.

Static weaving

Static weaving is implemented in two ways:

- Source code transformation
- Byte code transformation

In the source code transformation approach, Aspects are physically woven into the classes that make up the base application at compile-time of the application by source-code transformations. The core application source files, aspect source files are pre-processed by the aspect compiler to produce woven source code. This resulting source code is fed to the base language compiler to produce the executable. There is a variation of this approach which is used by the AspectJ tool. The aspect compiler instead of producing a woven source code, directly results in woven class files which can be executed by the Java Virtual machine.

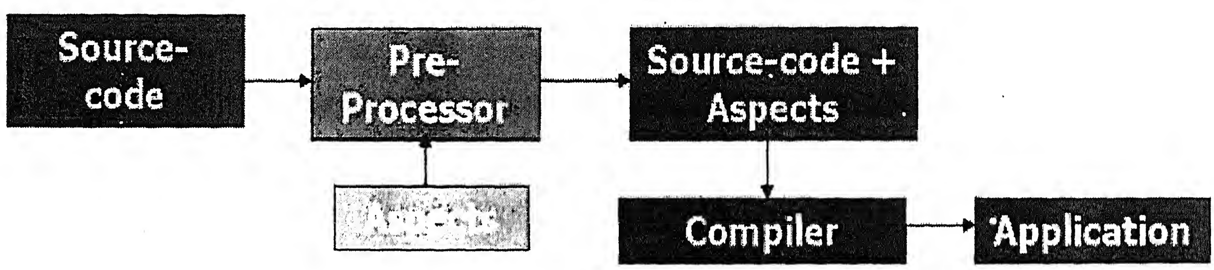


Figure 3.1: *Source code transformation*

In the byte code transformation approach, Aspects are physically woven into the classes that make up the base application at compile-time of the application by

byte-code transformations. The core application source files are initially compiled into class files by the base language compiler. These resulting class files are then fed as input to the aspect compiler. The aspect compiler then weaves the aspects into these class files producing the woven class files.

The main advantage of static weaving is that there is no run-time overhead. But the disadvantage is that application needs to be recompiled every time whenever aspects are added or removed.

Load-time weaving

Aspects are physically woven into the classes that make up the base application at load-time of the application. Load-time weaving is achieved by making use of a specialized class-loader that allows inserting aspect-code in the specific classes. This approach is used by JMangler, JavaAssist, AspectWerkz, JBoss/AOP. So, weaving is done in "just-in-time" style.

The advantage of this approach compared to static weaving is that aspects need not be weaved prior to the execution. Decision of which aspects to add can be deferred until load-time of the application. Application does not need to be recompiled to add or remove aspects. The disadvantages are the use of specialized class loader and run-time overhead at startup time of JVM..

3.3 Advice weaving in AspectJ

Before moving on to the mechanism of dynamic weaving, let's have a brief look at advice weaving in AspectJ.

The front-end of the AspectJ compiler is implemented as an extension of the Java compiler. The front-end compiles both AspectJ and pure Java source code into pure Java byte code annotated with additional attributes. These attributes represent any non-java forms such as advice and pointcut declarations.

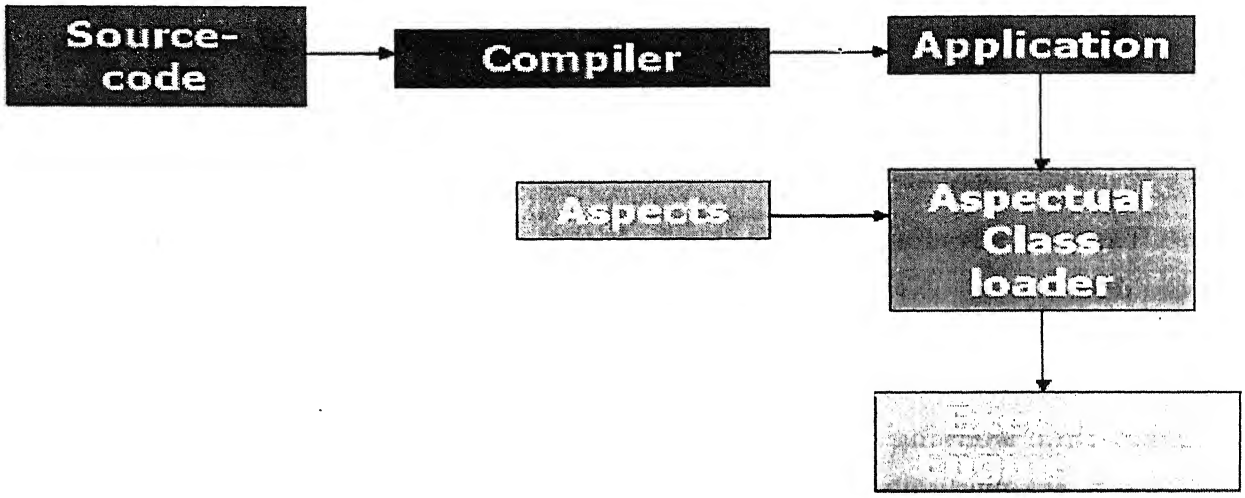


Figure 3.2: *Load-time weaving*

The back-end of the AspectJ compiler instruments the code of the system by inserting calls to the precompiled advice methods. It does this by considering that certain principled places in byte code represent possible join points; these are the "static shadow" of those join points. Each piece of advice is matched with each static shadow and checked if the advice's pointcut matches the static shadow. If a match is found, a call to the advice's implementation method is inserted, guarded by any dynamic testing needed to ensure the match.

Chapter 4

Dynamic weaving

Static weaving produces well-formed and highly optimized woven code whose execution speed is comparable to the code written without AOP. But still it has some serious drawbacks. Static weaving makes it difficult to identify aspect-specific statements in the woven code. Identity of the aspects is lost at runtime. As such modifying, inserting or deleting aspects during runtime can be expensive or not possible at all, even if the programming language used supports program manipulation through reflection interface.

The application needs to be recompiled each time, whenever aspects are added or removed. The decision to which aspect is to be woven has to be done beforehand, leaving no chance to reconfigure the system at runtime without aborting and restarting it. There is no mechanism to switch aspects basing on the runtime context. There are certain environments where it is needed to be able to change the global policies implemented through aspects during run-time. This is not possible with static weaving. As such, all the possible policies have to be embedded into the system, even though one is not enforced usually. This results in the tangling of the policies in an aspect.

Dynamic weaving means that aspects can be applied or removed at any time during runtime. Thus dynamic weaving allows the integration between components and the aspects at run time, resulting in a system which is more adaptable and

extensible. In short, in dynamic weaving aspects can be added to the system on the fly and, thus help avoid re-compiling, redeployment and re-start of an application.

4.1 Need for Dynamic AOP

There are various scenarios which call for dynamic weaving. For instance, one may have a huge resource-consuming image processing algorithm as part of an application. Depending on system load and available computing nodes, a trade-off between data distribution, the memory allocation scheme, and the utilization of computing power at runtime, has to be made. Both memory allocation and data distribution are crosscutting concerns, but the selection must be performed at runtime by the application. This necessitates dynamic AOP.

Another good example is that of an adaptable response cache in a web application. Response cache is a crosscutting concern and hence needs to be plugged as an aspect into the system. To make the response cache adaptable, the system should be able to switch the aspects depending on the runtime context (no. of clients, hit ratio, CPU usage..), wherein each aspect represents a particular policy or strategy. In case of a low hit ratio, one aspect should be used. In case of a marginal high hit ratio, some other aspect representing a policy should be used. This switching of aspects at runtime cannot be done using static weaving. In case of static weaving, all the available policies written as aspects should be weaved into the system beforehand. Whenever the execution reaches a potential join-point (potential means "requires advice to be executed"), all the available caches are to be checked whether it is turned on or off. This leads to high runtime overhead. If dynamic weaving is used, only the activated aspects can be woven thereby avoiding unnecessary runtime checks.

Various mechanisms for dynamic weaving of aspects have evolved and some tools have been developed in the recent years, such as PROgrammable extenSions of sErvices (PROSE)[12], Java Aspects Components (JAC)[13], Cross Language Aspect Weaver (CLAW)[4] etc. These systems give the programmer the ability to

dynamically modify the aspect code assigned to application join points. However, they offer a limited set of language join-points.

4.2 Dynamic weaving approaches

In general, implementations of aspect weavers are based on inserting hooks at a number of join points such as method calls, field modifications, field access etc. If the program execution reaches a join point, the inserted hook intercepts it and executes the advice if the reached join point is included in the list of join points identified by the pointcut.

The two most widely used approaches for dynamic weaving are:

- Interpreter approach
- Byte code manipulation

There is also another approach in use, "proxy approach [6]" though not as widely used compared to the former ones.

Proxy approach (Aspect Moderator framework)
In this approach, which has its roots from Concurrent Object-Oriented Programming, the architecture consists of 3 main components -

- Functionality Proxy
- Aspect Moderator
- Aspect Factory

The Aspect Factory performs the creation of aspects. The proxy intercepts the requests for the creation of aspects and method calls. When the proxy identifies a request for an aspect creation, it notifies the Aspect Factory of the request. Once the creation of the aspect is done by the Aspect Factory, proxy notifies the Aspect Moderator of an aspect insertion and registers the aspect with it. When the proxy

identifies the request for a method invocation, it notifies the Aspect Moderator of it, which then evaluates the aspects associated with the invoked method.

Interpreter approach

This approach [15] consists of an extension to the interpreter. A plug-in consisting of an API for inserting and unweaving aspects at runtime is added to the interpreter. The interpreter is extended to check for events occurring in the execution and on the occurrence of an anticipated event, advice is applied. This approach makes use of Java Virtual Machine Debugger Interface (JVMDI), which is one of the three interfaces of the Java Platform Debugger Architecture (JPDA).

JVMDI is an interface to inspect the state and control the execution of applications running in the Java Virtual Machine (JVM). JVMDI can be notified of occurrences through events such as breakpoint event, field access event etc. JVMDI can query and control the application through different functions in response to events. Using JVMDI, hooks are inserted as breakpoints at the join points. The advices, upon the arrival of the breakpoints during the execution are executed by the debugger. The advantage of this approach is that, hook insertion is fast. The drawback is that, there is a performance tradeoff due to the transfer of control from the application thread to the debugger thread. This approach is employed by tools such as PROSE, Axon.

Byte code manipulation

This approach consists of manipulating the byte code of the application classes. In this mechanism, hooks are inserted as method calls. The byte code of the class is modified, wherein the byte sequence representing the advice execution is embedded into the program. The runtime replacement of the byte code can be done by the hotswap mechanism of JPDA. Recently developed "Byte Code Engineering Library" can also be used for the same. In this approach, the advice execution is fast, though hook insertion takes time.

This approach is employed by tools like JAC, Wool etc. JAC uses the Javassist load-time tool to transform the base objects for making them capable to be dynamically wrapped. The drawback in JAC is that the hooks are inserted at all join points, even at those which are not potential join points, resulting in poor performance.

Chapter 5

Dependency Problems

Dynamic aspect weavers provide a solution to the problem of switching off any aspects, modifying already added ones or inserting any new ones at runtime. The drawback with currently available static and dynamic weavers is that there is no support for specifying the dependency of the aspects. Though some support is available in AspectJ, it is limited in its capabilities. PROSE, a dynamic aspect-weaving tool, executes aspects in the reverse order of their insertion. Even in the case of a single aspect, the advices concerning a single join point are executed in the reverse order of their specification.

According to Huang [7], aspects not only interact with the primary abstraction but also with other aspects. Even in a single aspect, a dependency can exist between the after and before advices. Since weave order cannot be specified, weaving multiple aspects at a time or even a single aspect, deviating from the dependency order would lead to problems in the system. Failure to preserve dependency order of aspects would lead to inconsistencies in various states of the system. These undesirable behavior is termed "emergent properties".

5.1 Dependencies

According to Kienzle [10], aspect dependencies are categorized into three kinds:

- Orthogonal
- Uni-directional
- Circular

Orthogonal aspect provides functionality, which is independent of the functionalities of the other aspects or the base application. The only thing it depends on is activation time or any application-independent information provided by the run-time environment, e.g. current method name, etc. For example: logging is an orthogonal aspect. A logging aspect can be applied to various places in an application to print out stack traces.

Uni-directional aspect provides functionality that is dependent on the functionalities of other aspects in the system. Uni-directional aspects are further classified into two types:

- Uni-directional preserving
- Uni-directional modifying

Uni-directional preserving aspect does not affect the functionality of any other aspect. Ex.- Billing aspect depending on the timing aspect. For example, consider a telecom application. To set up correct billing, the elapsed time of long distance phone calls must be measured. The long distance timing aspect uni-directionally depends on the call aspect. It is not orthogonal, because it has to associate timing with calls, and therefore depends on the call aspect. The billing aspect in turn depends on the call and the timing aspect. The reason is, it has to know the calls source and destination city, and the elapsed time to calculate the total cost.

Uni-directional modifying aspect modifies the functionality of some part of an application, but transparently without the latter's awareness. In a sense, a uni-directional modifying aspect wraps around or encapsulates some services provided by other aspects. As a result, some of the original services might not be provided

anymore. For example, consider a typical banking application which allows good clients to overdraw their account. Clients with a bad credit history on the other hand are not allowed to do this. The desired effect can be achieved by encapsulating in one aspect the account behavior, and design an additional aspect that denies withdraw requests in case of insufficient funds. The additional aspect removes the withdraw service from the account aspect.

Circular dependency exists when several aspects form a chain of mutual dependencies among them.

5.2 Consistency Problems

Essentially, by dependency we mean a write operation followed by a read or a write followed by a write. We can see that orthogonal aspects do not pose any concern with regard to the dependency order issue, but others do. The weaving and execution of aspects can be in a single thread of execution, i.e. no concurrent executions. It can also be done through multiple threads of execution. However, the consistency problems that we will be discussing will be same in both the cases.

We classify the inconsistencies, which occur due to dependency-failing-aspect-executions in a fashion similar to the levels of consistency in the database terminology like "Repeatable read", "Read committed", and "Read uncommitted".

Repeatable read allows only committed records to be read. It also requires that, between two reads of a record by a transaction, no other transaction is allowed to update the record.

Read committed is more relaxed compared to repeatable read in the sense that between two reads of a record by a transaction, the records can be updated by any other committed transaction.

Read uncommitted is the lowest level of consistency. It allows even uncommitted records to be read.

When we consider the problem of inconsistency caused due to the failure in preserving aspect dependencies during execution, the notion of "problem" depends on the level of inconsistency acceptable by the user and the domain under consideration. For example - consider an application, where the user queries for search results on the web during the execution of the application. The user may well be comfortable with the partial results or he may as well be less bothered even the results change for two consecutive searches. In that case, we can say that the user allows for weaker levels of consistency.

The upcoming discussion deals with the before and after advice declarations. Advice defines the additional code that should be executed at the join points. 'Before advice' is executed when a join point (say field modification join point) is reached. When the control returns from the join point, 'after advice' is executed. The problems illustrate only generic instances, and do not deal with any real time domain problems.

The discussion assumes that the weaving follows the ordering as in tools like PROSE, JAC etc. That is, if there are 'm' aspects woven, the execution order goes as:

$\boldsymbol{B_1\ B_2\ B_3\ \ldots\ B_m\ F\ A_1\ A_2\ A_3\ \ldots\ A_m}$
B - before advice; **F** - base functionality; **A** - after advice.

Some tools like AspectJ weave in this fashion:

$\boldsymbol{B_1\ B_2\ B_3\ \ldots\ B_m\ F\ A_m\ \ldots\ A_3\ A_2\ A_1}$

We base our discussion on the former. The analysis for the latter can be done on similar lines.

If we are weaving at runtime, the *program counter* could be at any point along this sequence at the time of weaving. This gives rise to different kinds of problems that we will discuss further. We assume that the developer of an aspect has prior knowledge of the aspects woven into the system. Therefore, the developer while providing the aspect for weaving at runtime specifies its dependencies with the existing aspects.

5.2.1 Static weaving

We begin our analysis with the static case, observe the dependency problems and move on to dynamic weaving. As said before, AspectJ facilitates for specifying aspect priorities. However, it is limited in its capabilities. Giving a fixed order to the weaving of aspects (like PROSE), would lead to inconsistent states in the system.

Now, we will try to analyze the various dependencies that need to be considered in the weaving of aspects.

We use the following notation to specify the dependencies:

$D(A_i, B_j)$: There is a dependency of A_i on B_j. A_i reads/writes a value written by B_j

$$B_j \longrightarrow A_i$$

Figure 5.1: *Dependency of A_i on B_j*

Whenever we say 'not solvable' in the future discussion, we mean that an ordering, satisfying the constraints cannot be obtained.

Problems with static weaving

There are two cases to consider:

- **Case1:** If a single aspect is to be woven, the order of weaving the aspect is not concerned by whether it's after advice has a dependency on the before advice or not. The final weave order would be: $B_1\ F\ A_1$
- **Case2:** If two or more than two aspects are to be woven, and if dependencies exist among them, then those dependencies have to be analysed as to whether a valid ordering satisfying these constraints can be obtained or not. For the moment, we consider only two aspects.
  - If the aspects are orthogonal, they can be woven in any order.
  - If the aspects are non-orthogonal and some of them are uni-directional, then we need to consider the dependencies specified among them to obtain a valid ordering.

    We assume that by default, the weaving order is $B_2\ B_1\ F\ A_2\ A_1$

    The dependencies between two aspects asp 1 and asp 2 can be any of the below specified ones:
    $D(A_1, B_1)\ D(A_2, B_2)\ D(B_1, B_2)\ D(B_2, B_1)$
    $D(A_1, B_2)\ D(A_2, B_1)\ D(A_1, A_2)\ D(A_2, A_1)$

    We left out $D(B_i, A_j)$, as we assume that before advices do not depend on the after advices.

    If $D(A_2, A_1)$ or $D(B_2, B_1)$ or both is/are specified, then the weave order would be in contrast with the required. Violating the dependency order would lead to **read committed and read uncommitted** inconsistencies. The corresponding advices have to be swapped to maintain the dependency.

Similarly, we can easily state that if a cyclic dependency occurs with the specified dependencies, then the aspects cannot be woven. Swapping would not solve the problem.
Ex:- $D(A_1, A_2)$ and $D(A_2, A_1)$; $D(B_1, B_2)$ and $D(B_2, B_1)$

No other combinations of the above-mentioned dependencies would lead to problems. The reason is that of the above eight dependencies, four are of an after advice depending on a before advice. These dependencies whether present or not would not be of any problem. The reason is that after advices always come later in execution relative to the before advices.

Of the remaining four dependencies, the dependencies involving only before advices would not lead to any inconsistency in the after advices. Therefore, only a cyclic ordering among the dependencies as specified before, would lead to an unsolvable problem.

In case of a possible weave ordering based on the dependencies, the order of the advice execution would be any of the below:
$B_2\ B_1\ F\ A_2\ A_1$
$B_1\ B_2\ F\ A_1\ A_2$
$B_1\ B_2\ F\ A_2\ A_1$
$B_2\ B_1\ F\ A_1\ A_2$

In the case of more than two aspects to be woven into the system, the analysis would be the same with the same problems occurring, as specified above.

5.3 Dynamic weaving

Before going into the discussion, we assume that a valid ordering is obtained for the statically woven aspects. Now coming to dynamic weaving, we strive to obtain an order equivalent to that of the static case i.e.

If $D(X, Y)$ is required: Weave Y before X.

If we consider an aspect that is already woven and at runtime, a new aspect or a modified version of the same aspect is to be woven, there are different weaving points depending on when the aspect arrives. Since the weaving command could come at any arbitrary time, the points at which weaving can be done, are as shown below:

Ex:- Advice order - $\boldsymbol{B_1}$ $\boldsymbol{F}$ $\boldsymbol{A_1}$

If an aspect is to be woven at runtime, the points to be considered are:

$$\uparrow \quad B_1 \quad \uparrow \quad F \quad \uparrow \quad A_1 \quad \uparrow$$

The arrows represent the possible points for weaving.

Therefore, in dynamic weaving, two issues come into consideration. One is, whether a valid weave order is possible for the given dependencies. If so, can the weaving be done immediately? There will be instances where weaving can be done, but not at the very point of arrival of the aspect. Weaving of the aspect may need to be deferred to a later stage in execution. All this we will see in our coming subsections.

We have considered three types of scenarios to present the dependency order problem. They will be dealt in succession.

5.3.1 No pre-existing aspects

In our first scenario, we assume that no aspects are woven statically. An aspect *asp1* is to be woven at runtime into the system.

In this situation, the aspect can or cannot be woven directly at the very instance of arrival. It depends on whether the after advice has a dependency on the before advice or not. It also depends on the point of execution at which the aspect arrived.

If the aspect arrives before the execution of the join point functionality, then the aspect can be immediately woven.

If the aspect arrives after the execution of the join point functionality, then the dependency of the after advice on the before advice has to be considered. If dependency exists, then weaving should not be performed at that instant. It should be deferred to a later stage in execution. Otherwise, weaving can be performed.

Therefore, the final advice execution order would be:
B_1 F A_1 (if aspect arrives before 'F')
or
F A_1 (if aspect arrives after 'F' and weaving is allowed)
or
F (if aspect arrives after 'F' and weaving is deferred)

5.3.2 One aspect already exists

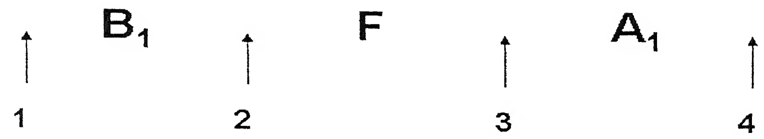
In this scenario, we assume that we already have an aspect *asp1* woven into the system at static or runtime. A new aspect *asp2* is to be woven. Here two cases need to be considered.

If *asp2* is orthogonal, it can be woven into the system. However, decision should be taken as to whether it can be woven immediately. The analysis required is the same as that considered in the previous section. It is not restated to avoid redundancy.

If *asp2* depends on the services of the *asp1*, then the dependencies specified should be considered before making the decision.

Advice order before weaving: B_1 F A_1

The new aspect may arrive at any of the four weaving points shown below:



In the below analysis, each case refers to a weaving point.

- **Case1:** If *asp2* is to be woven before the execution of B_1, then it can be directly woven, if a valid ordering satisfying the inter-aspect dependencies is obtained. For this, the analysis to be done is the same as that done in the case of static weaving, where two or more aspects are to be woven.

  The final advice execution order can be any of the following:
  B_2 B_1 F A_2 A_1
  B_1 B_2 F A_1 A_2
  B_1 B_2 F A_2 A_1
  B_2 B_1 F A_1 A_2

- **Case2:** Initially, we check whether, a valid ordering satisfying the inter-aspect dependencies can be obtained or not. If not, weaving of the aspect should be done.

  If a valid ordering can be obtained, it should be checked whether $D(B_1, B_2)$ exists.

  - If $D(B_1, B_2)$ does not exists, then weaving can be done.
  - If $D(B_1, B_2)$ exists, then weaving is attempted where execution of B_2 doesn't happen. Now, it needs to be checked whether $D(A_2, B_2)$ exists or not.

    * If $D(A_2, B_2)$ does not exists, then weaving can be performed. At the time of execution, both B_2 and A_2 would get executed.
    * If $D(A_2, B_2)$ exists, weaving the aspect immediately would lead to read uncommitted inconsistency as A_2 would access data which should have been written by B_2 (which does not get executed). In such a case, weaving should be postponed to a later point in execution, where the dependency can be satisfied.

  The final advice execution order can be any of the following:

  $B_1\ B_2\ F\ A_2\ A_1$

  or

  $B_1\ F\ A_2\ A_1$

  or

  $B_1\ F\ A_1\ A_2$

  or

  $B_1\ F\ A_1$

- **Case3:** In this case, its for sure that B_2 doesn't get executed, if the aspect is weaved. Therefore, the analysis should be done by considering the dependence of A_2 on B_2.

  Initially, we check whether, a valid ordering satisfying the inter-aspect dependencies can be obtained or not. If yes, we examine whether $D(A_2, B_2)$ exists or not.

  - If $D(A_2, B_2)$ does not exists, then weaving can be performed, as A_2 doesn't require any state from the unexecuted B_2.
  - If $D(A_2, B_2)$ exists, weaving should not be done immediately. Weaving the aspect immediately would lead to read uncommitted inconsistency as A_2 would access data written by B_2 (which does not get executed). Weaving of the aspect is deferred to a later stage where the dependency is satisfied.

  If a valid ordering cannot be obtained with the inter-aspect dependencies, weaving should not be done.

The final advice execution order can be any of the following:

$B_1\ F\ A_2\ A_1$

or

$B_1\ F\ A_1\ A_2$

or

$B_1\ F\ A_1$

- **Case4:** This is the same as that of Case3, except for a small relaxation. The weaving point is after the execution of A_1.
  - If $D(A_1, A_2)$ exists, the new aspect can be weaved immediately, as none of B_2 and A_2 would be executed. So, there would be no problems.
  - If $D(A_1, A_2)$ does not exists, A_2 would come after A_1 in the advice ordering. Now the scenario is same as Case3. It is not repeated to avoid redundancy.

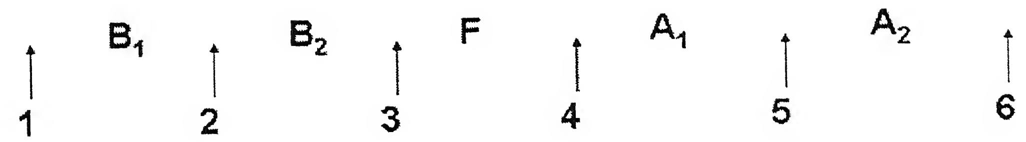
Issues concerning the weaving of multiple aspects at a time involve considerations of multiple issues specified above. What can be stated is that concerns involving multiple aspects can be resolved into issues involving two aspects, one aspect at a time.

5.3.3 Insertion of a modified aspect

In this scenario, we assume that two aspects *asp1* and *asp2* are already woven into the system and are in execution. We need to examine the situation when one of the aspects changes. That is, it is modified and sent to be re-woven. This involves the withdrawal of the older version of the aspect. In such a situation, one should consider the dependencies lost due to withdrawal. For the moment, we won't consider the withdrawal dependencies. We would be dealing them later. We base our discussion with the insertion of re-versioned *asp1.*

Before weaving, advice order: $B_1\ B_2\ F\ A_1\ A_2$

The weaving points to be considered are:



In the following analysis, each case refers to a weaving point.

- **Case1:** If a valid ordering can be formed from the inter-dependencies involving *asp2* and $\boldsymbol{asp1'}$, *asp1* should be removed and $\boldsymbol{asp1'}$ be inserted. The analysis to be done to determine the possibility of weaving is the same as that done in the case of static weaving, where two or more aspects are to be woven.

  The final advice execution order can be any of the following:
  $\boldsymbol{B_2\ B'_1\ F\ A_2\ A'_1}$
  $\boldsymbol{B'_1\ B_2\ F\ A'_1\ A_2}$
  $\boldsymbol{B'_1\ B_2\ F\ A_2\ A'_1}$
  $\boldsymbol{B_2\ B'_1\ F\ A'_1\ A_2}$

- **Case2 - 4:** In this scenario, older version B_1 would be executed. So, if we weave modified aspect $\boldsymbol{asp1'}$, $\boldsymbol{A'_1}$ (re-versioned A_1) would get executed. This may not be feasible if $\boldsymbol{D(A'_1, B'_1)}$ exists. To weave the newly modified aspect $\boldsymbol{asp1'}$, dependence of $\boldsymbol{A'_1}$ on $\boldsymbol{B'_1}$ should be checked.
  Initially, we check whether, a valid ordering satisfying the inter-aspect dependencies can be obtained or not. If yes, we examine whether $\boldsymbol{D(A'_1, B'_1)}$ exists or not.

  - If $\boldsymbol{D(A'_1, B'_1)}$ doesn't exists, weaving can be done.
  - If $\boldsymbol{D(A'_1, B'_1)}$ exists, weaving of the aspect should not be done immediately. Weaving the aspect immediately would lead to read uncommitted inconsistency as $\boldsymbol{A'_1}$ would require state/data from $\boldsymbol{B'_1}$ (which does not

get executed). Therefore, weaving of $asp1'$ must be deferred to a later stage in execution.

In case, weaving cannot be done or deferred to a later execution point, executions with the older versions will continue.
The final advice execution order can be any of the following:
$B_1\ B_2\ F\ A_2\ A_1'$
or
$B_1\ B_2\ F\ A_1'\ A_2$
or
$B_1\ B_2\ F\ A_1\ A_2$

- **Case5 - 6:** Already the advices concerning the join point have been executed. So, weaving can be done if a valid order can be obtained from the aspect dependencies specified.
  The final advice execution order would be:
  $B_1\ B_2\ F\ A_1\ A_2$

The analysis is similar in case of weaving the modified version of *asp2*.

Similarly, the scenario of a single orthogonal modified aspect to be rewoven is a special case of this section.

There may be some domain specific situations, where the version difference between advice executions may not be of problem.. In such a situation, in Case 2 - 4, modified aspect can be woven though $D(A_1', B_1')$ exists and B_1' is not executed.

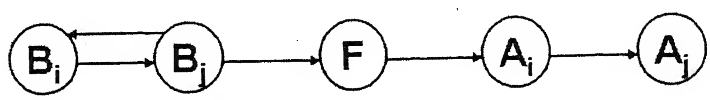
5.4 Graph formalism

The dependency analysis can be represented in a graph theoretic framework. In this framework, each functionality (join-point functionality/advice) is represented as a node.

If we consider the static case of aspect weaving, a topological sort of the nodes is done, where the nodes represent the advices and the join-point functionality. The

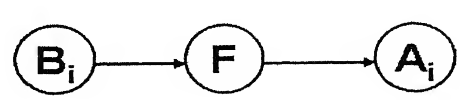
topological sort is done basing on the dependencies of the advices. For example: if A_j has a dependency on A_i, A_j comes latter in order with relative to A_i. If there exists a cycle in the graph ordering, weaving cannot be done. After advices are implicitly dependent on the join-point functionality, which itself is dependent on the before advices.

The below figure illustrates the case of a weave ordering where cyclic dependency exists and weaving cannot be done.

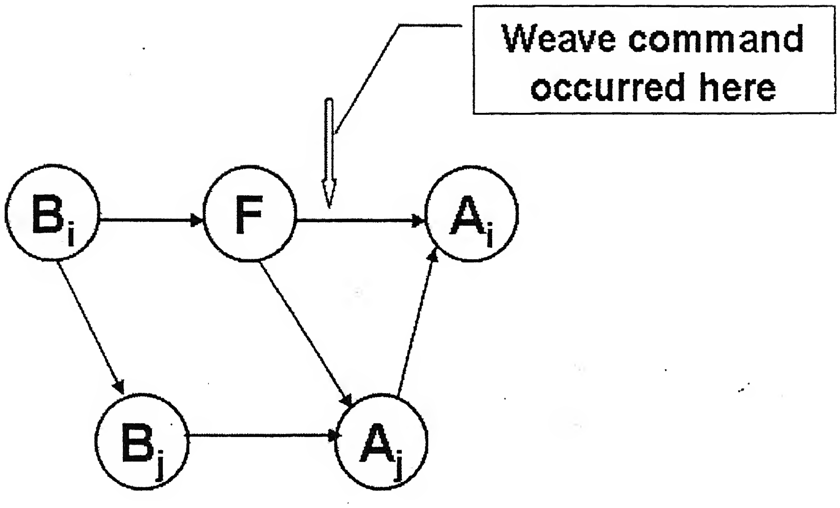


In case of dynamic weaving, the point at which the insertion command occurs should also be considered. Initially we form a graph with the dependencies specified. Next, depending on the weave point and the dependencies, we knock out nodes which do not fit in the graph. This is done as per the analyses given in the previous section. If a node is removed, all its dependent nodes are also removed. We then form a topological sort of the nodes remaining in the graph. This gives the advice order at that instant. As the execution proceeds, nodes are added, when the dependencies are satisfied. This is illustrated below.

Initial advice execution order:



An aspect is to be inserted, where the after advice depends on the before advice. The weave command occurred after the execution of the join-point functionality. Below is the graph drawn with the dependencies specified.



We know that the advices B_j and A_j cannot be weaved, as B_j doesn't get executed and A_j which is dependent on the former, executes thereby leading to inconsistency. So, we knock out node B_j from the graph. Then we remove the nodes dependent on B_j, and form the topological sort of the nodes thus remained. Of course, in this example, the graph remains the same. In case, $D(A_j, B_j)$ doesn't exists, node A_j will remain in the graph, with node A_i following it after the sort.

5.5 Some supplements

In the previous sections, we have considered the problem of dependencies and to when an aspect is to be weaved with regard to a single join point. In general, an aspect may specify advices for multiple join points. In such a case, it may happen that a valid ordering is obtained for some join points and not for others. It may also happen that, the aspect can be weaved immediately for some join points and should be deferred for others.

So, in case of static weaving, an aspect should be weaved only when a valid advice ordering is obtained for all the join points. In case of dynamic weaving, an aspect should be weaved only when a valid advice ordering is obtained for all the

join points and it should be weaved at all the join points at the same instant.

As said before, withdrawal of an aspect would have implications to other aspects. There may be some aspects dependent on the aspect to be removed. Removal of such an aspect would lead to inconsistencies in the system. So, withdrawal of an aspect should not be done if there are aspects dependent on it.

There is another important issue to consider. We shall illustrate this by an example. Let's say for example, a method 'A' called another 'B' which in turn invoked 'C'. Suppose that, an "insertion of aspect" command is given at the moment 'C' is under execution. Suppose that the aspect has both "before" and "after" advices, wherein the after advice depends on the before advice. Also, assume that, the aspect specifies advices for method execution join points where the method name is 'A' or 'B'. Now, the aspect has no bearing for method 'C' which is under execution. but that doesn't mean that we can weave the aspect immediately. The reason is that after the completion of method 'C', control goes back to 'B'. If we had weaved the aspect during the execution of 'C', the after advice of the aspect which gets executed upon exit of 'B' would get invalid state/data due to non-execution of the before advice upon entry of 'B'. Similar is the case with method 'A'. So, the aspect should be weaved only after the exit of method 'A'.

So, an inspection of the stack trace should be done before insertion of an aspect. Insertion should be done, only when the deepest (in the stack) potential join point corresponding to the aspect has executed. Of course, this problem does not occur, if the aspect to be woven has no before advice.

Chapter 6

Implementation

A dynamic aspect weaver requires information regarding the dependencies among the aspects. The class constructs written for the tool should facilitate for the specification of dependencies (as in the case of AspectJ that provides "declare precedence" keyword).

Once the dependencies are specified, the analyses discussed in the previous chapter, should be done based on the run-time information for dynamic weaving. For this JVMDI can be used to obtain information of the stack frame and method executions etc. during run-time.

When an aspect cannot be woven, feedback is provided to the aspect developer specifying the dependent aspects. The system will continue its operation with the woven aspects.

In general, two approaches are used for the development of a dynamic aspect weaver. They were already explained in the fourth chapter. It is repeated again for convenience. In the first approach, hooks are inserted as breakpoints, which are handled through the Java virtual machine debugger interface (JVMDI). JVMDI is one of the three interfaces of the Java Platform Debugger Architecture (JPDA), designed for use by debuggers in development environments. JVMDI is an interface to inspect the state and control the execution of applications running in the Java

Virtual Machine (JVM). The JVMDI client can be notified of interesting occurrences through events such as Breakpoint event, Field event etc. JVMDI can query and control the application through many different functions, either in response to events or independent of them. In this approach, there is a performance tradeoff due to the transfer of control from the application thread to the debugger thread.

In the second approach, hooks are inserted as method calls. The byte code of the class is modified, wherein the byte sequence representing the advice execution is embedded into the program. The runtime replacement of the byte code can be done by the hotswap mechanism of JPDA. "Byte Code Engineering Library" can also be used for the same. In this approach, the advice execution is fast, though hook insertion takes time.

For the development of our weaver, we adopted the approach of inserting hooks as breakpoints. If an aspect were to be removed or modified, modification of byte code as in the case of second approach would be expensive and would lead to performance loss. The process would just suffice to removing the breakpoints at the join points, in case of the first approach.

6.1 Features

The weaver is written in java, obtaining the advantage of state inspection by JVMDI. Using JVMDI, execution of applications can be controlled by intercepting the following events:
Field Access event
Field modification event
Method Entry event
Method Exit event
Exception Throw event
Exception Catch event
Class Load event
Class Unload event

Thread Events
Frame Pop event

Our implementation facilitates specification of aspect advices only for the method entry/exit, field access/modification, exception throw/catch events. The syntax for representations and the working mechanism is adopted from PROSE.

An aspect consists of one or more crosscuts. Each aspect defines a method *isModified( )* retuning true/false depending on whether it is a new aspect or a modified version of an already inserted one. In case of a modified aspect, the method *modificationOf( )* returns the aspect object which is modified.

Consider an application to be traced for field access events. Below is an example of a tracing aspect:

Listing 6.1: An example aspect

```
import weaver.*
public class ExampleAspect1 extends Aspect
{
  Crosscut1 myCrosscut1 = new Crosscut1();
  Crosscut2 myCrosscut2 = new Crosscut2();
  public Crosscut[] crosscuts()
  { return new Crosscut[]{myCrosscut1,myCrosscut2};}
  public boolean isModified()
  { return false;}
  public Aspect modificationof()
  { return null;}
}
```

A **crosscut** defines the join-points, advice and in case of dependency, the crosscut on which it depends. It may happen that, a new aspect is to be inserted into the system with the presumption that, an already inserted aspect depends on it.

This can be done by re-writing the existing aspect(dependant) specifying the new dependency and weaving both the aspects. This would result in performance loss. For this purpose, a method returning the dependants of an aspect is provided. A crosscut is specified as below:

Listing 6.2: An example crosscut

```
import weaver.*;
public class Crosscut2 extends GetCut
{
  public static void GET_ARGS(Appl x, int y)
  { System.err.println("advice:"+y+"accessed");}
  protected PointCutter pointCutter()
  { return  Fields.modifier(2);}
  public Crosscut[] depends()
  { Crosscut1 eCross = new Crosscut1();
    return new Crosscut[]{eCross};
  }
  public Crosscut[] dependants()
  { return null; }
}
```

The join-points to which the crosscut applies are defined by the parameters of the GET_ARGS method. In the above example, the advice is applied on the access of a field defined in a class "Appl" and of type "integer". The arguments change basing on the event, which is to be trapped. A crosscut defined for a field access event extends the "GetCut" object which is a Java class defined by the weaver. For such a crosscut, the first parameter is the receiver type. Second parameter is the value of the field. Given below is the list representing the arguments of the advice methods corresponding to each type of join point.

There is also a facility to specify receiver of any type using "ANY". Similarly, in case of method events, a set of parameters of different types can be specified using

Event	Parent	Advice method	Arguments
Method entry/exit	MethodCut	METHOD_ARGS	Receiver type, Method arguments
Field access	GetCut	GET_ARGS	Receiver type, Field value
Field modification	SetCut	SET_ARGS	Receiver type, Modified Field value
Exception catch	CatchCut	CATCH_ARGS	Exception type
Exception throw	ThrowCut	THROW_ARGS	Exception type

Table 6.1: Advice parameters for each join point

"REST". For example, if the method has three arguments - int x, string str, int y : advice call for such a method in any class is written as METHOD_ARGS(ANY a, int x, Rest r).

The possible set of join-points for a crosscut can be further reduced by specialization. This is achieved by using filters. In the above example, the advice is applied on the access of a field which is "private". Some examples of filters are Method.name("meth1"), Exception.occuredinClass(Appl).

6.2 Architecture & Mechanism

The functionality of the tool can be broadly divided into two parts. One is the insertion and deletion of aspects. The second is the registration of events for notification from the JVM and execution of advices upon being notified by JVM. Our implementation provides an API for specifying the aspects, crosscuts, join-points etc. The modules for the insertion/deletion of the aspects and the API for specifying the aspects by the user are written in Java, taking into consideration the use of JVMDI to inspect the state during execution. The mechanism for registration of events for notification, setting of breakpoints, obtaining information about the thread stack during execution is implemented in native language, C. The interaction between the java and native implementations is done through JNI.

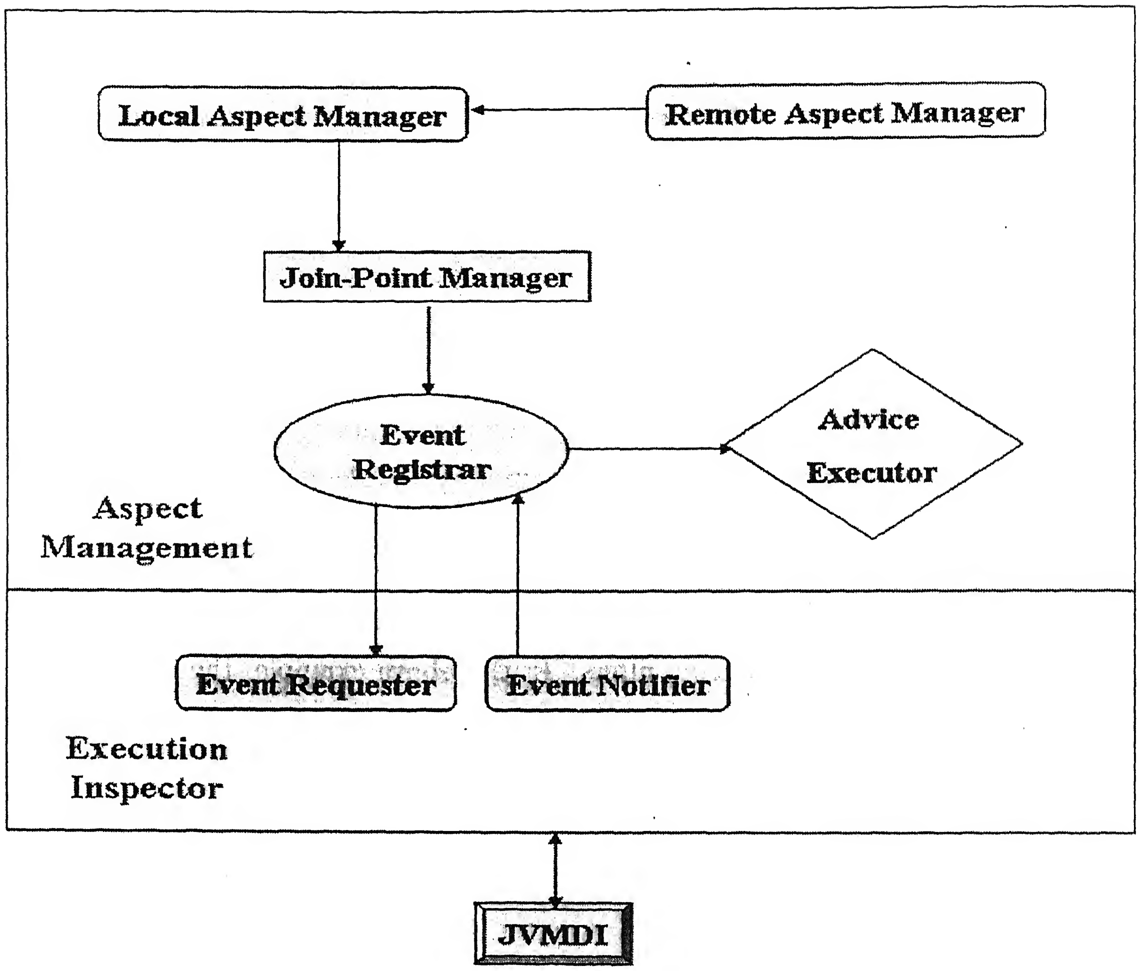


Figure 6.1: *Architecture - module view*

Aspect management module deals with the insertion/withdrawal of aspects, registering of crosscuts pertaining to their join-point requests. It interacts with the ***Execution inspector*** to register the event requests for notifications with the JVM. We would be discussing briefly on the working of the various components of the aspect management module.

When an aspect is to be inserted at compile time, ***Local aspect manager*** initially checks if it is already inserted and not intended to be reinserted. In such a case, no insertion is done. Willingness to reinsert an aspect is specified using the

isModified( ) method, wherein the to-be modified aspect would be the same as the aspect to be inserted.

After the above process, ***Local aspect manager*** does dependency checks to determine whether inter-aspect dependencies can be satisfied. If not, insertion of the aspect is aborted and a feedback is provided to the user specifying the aspects whose dependencies are violated. In the case of dynamic weaving, if the dependencies can be satisfied at a later stage, the user application is continued to run until a point in execution is reached where insertion of the aspect can be done. The remaining process would be the same in case of both static and dynamic weaving.

In case an aspect is to be re-inserted, the existing one is removed. As said before, an aspect may contain one or more crosscuts, where each crosscut represents a set of join-points and the advice to be executed at these join-points during execution of the application. Now for each crosscut of the to-be inserted aspect, join-point requests are created.

For example, given below is a part of the body of a crosscut

```
public static void GET_ARGS(Appl x, int y)
{
  System.err.println("advice:"+y+"accessed");
}
protected PointCutter pointCutter()
{
  return Fields.modifier(2);
}
```

The above crosscut represents an advice for a field access event. For this crosscut, potential classes in the JVM that are applicable for this crosscut are determined. By potential, we mean those classes, whose type is same or derivable from the receiver type specified as the first argument of the advice method. Then, for each

of these potential classes, field access requests are created for each of its declared fields satisfying the type of the second argument of the advice method. In case of a method entry request, methods whose arguments satisfy the parameter list specified in the advice call are considered.

Once the corresponding event requests are created, they are filtered basing on the specialization defined in the *pointCutter( )* method. For the above example. field access requests where the field is "private" are considered.

After the join-point requests have been obtained, ***Local aspect manager*** invokes ***Join-point manager*** for the registration of the join-point requests with their corresponding crosscuts. In a sense, the crosscuts would be the listeners for their corresponding requests. ***Join-point manager*** maintains a listener list for the listeners(crosscuts) of each request and then invokes ***Event registrar*** passing it the listener list along with the object (method/field/exception object) for which join-point requests were created.

Event registrar maintains a hash map mapping each object to its listeners. It invokes the ***Event requester*** component in the ***Execution Inspector*** module to register events for notification from the JVM.

Event requester makes JVMDI calls to enable event notification for each of the event requests. Whenever a method/field/exception event occurs in the JVM. a notification occurs from the JVM. ***Event notifier*** then notifies the event to the ***Event registrar*** passing it the object (method/field/Exception object) involved in the event. ***Event registrar*** then obtains the list of listeners (crosscuts) interested in that corresponding object, from the hash map. This list is passed to the ***Advice executor***; which then calls the advice of each of the crosscuts in the list.

In case of insertion of an aspect at run-time, ***Remote aspect manager*** is involved. After getting the necessary parameters (name, location) of the aspect.

it invokes ***Local aspect manager*** and the process would then be the same as described earlier.

Withdrawal of an aspect works in similar lines as of insertion. In this case, ***Local aspect manager*** obtains the crosscuts of the aspect to be deleted and invokes ***Join-point manager*** passing each of the crosscuts. ***Join-point manager*** then unregisters each crosscut, i.e. it removes that crosscut from each of the listener list in which it is present. In case of a listener list being empty after withdrawal of an aspect, ***Event registrar*** disables the event notification for that object by invoking ***Event requester***, which makes JVMDI calls to remove any hooks inserted for that event.

Chapter 7

Conclusions

Aspect-oriented programming is a relatively new paradigm with the purpose of lowering complexity inherent to software development. Essentially, AOP "captures" multi-class interactions scattered throughout code (cross-cutting concerns) and encapsulates them into a single, centralized, class-like unit called an "aspect".

Right from the development of AspectJ, many tools have been developed over the few years addressing both static and dynamic aspect weaving. But these tools do not consider the dependencies among aspects before weaving them, which leads to inconsistencies in the system.

We presented an analysis that is necessary before an aspect can be woven at runtime. Essentially one needs to check whether a weaving is permissible, and if yes, determine the point of execution at which weaving can happen. We considered three types of scenarios to illustrate the analyses. From the analyses, we can identify that weaving can be done immediately (if it can be done), if the aspect arrives before the execution of any of the existing advices corresponding to the join-point. In the other cases, dependencies have to be considered. We developed a dynamic aspect weaver incorporating our analysis.

There are still some issues to be considered in dynamic weaving of aspects. In the current implementations, at run-time, join point checks are done to determine

if the advice needs to be executed on reaching a join point. In effect most of these fail causing a substantial run-time overhead. Before we see an example illustrating this fact, we define *join point shadows* as the locations that represent join points during runtime. We refer to shadows whose join points always lead to an execution of aspect-specific code as *unconditional join point shadows*, and those whose join points lead only under some circumstances to aspect-specific behavior as *conditional join point shadows*. For *conditional shadows*, the weavers add runtime checks determining whether or not aspect-specific code needs to be executed. If such a check succeeds, the aspect-specific code is executed.

For example, a developer wants to trace the control flow starting at a certain point in the execution of a program. Suppose there is a method 'A' whose code contains an unconditional call to method 'B' followed by a conditional call to method 'C'. At the same time, there are other control flows in the application using the methods 'B' and 'C'. Now, the developer wants to trace the method executions in the control flow of method 'A'. In general, the current implementations perform run-time checks on the execution of the method 'C', to determine whether it is being executed in the control flow of method 'A' or not. If there are a large number of different control flows using method 'C', most of the run-time checks will fail causing run-time overhead. The problem is that, it is not known at weave-time which methods would be invoked in the control flow of 'A'. Unless the condition is satisfied, method 'C' would not be executed in the context of method 'A'.

The above problem arises due to the fact that current dynamic weavers adopt the approach of *complete weaving*. By *complete weaving*, we mean determination of join point shadows and creation of join point checks is done beforehand. Once an aspect is weaved, set of join point shadows associated to an aspect is fixed and does not change at runtime. In more complex applications, complete weaving can lead to a huge number of adapted join point shadows whose join point checks fail and just produce runtime overhead. Solving this problem requires an analysis of the dependencies among the join points. This would be a good direction to extend our weaver.

Bibliography

[1] ALONSO, POPOVICI, AND GROSS.T. *Dynamic Weaving for Aspect Oriented Programming.* In1st Intl. Conf. on Aspect-Oriented Software Development, Enschede,The Netherlands, April 2002.

[2] ASPECTWERKZ. *Web site http://aspectwerkz.codehaus.org.*

[3] BOBROW, G.KICZALES, AND DES RIVIERES.J. *The Art of Meta-Object Protocol.* MIT Press 1991.

[4] CLAW. *Web site http://iunknown.com.*

[5] COMPOSITIONFILTERS. *homepage http://trese.cs.utwente.nl/oldhtml/composition_filters.*

[6] CONSTANTINIDES, T.ELRAD, M.E.FAYAD, AND P.NETINANT. *Designing an aspect-oriented framework in object-oriented environment.* ACM Computing surveys, March 2000.

[7] HUANG.J. *Experience Using Aspectj to implement Cord.* OOPSLA 2000.

[8] KICZALES,G., E.HILSDALE, J.HUGUNIN, M.KERSTEN, J.PALM, AND W.G.GRISWOLD. *Getting started with Aspectj.* Communications of the ACM, October 2001.

[9] KICZALES, J.LAMPING, A.MENDHEKAR, C.MAEDA, C.LOPES, J.LOINGTIER, AND IRWIN. *Aspect-Oriented programming.* In 1997. European Conf. on Object-Oriented Programming (ECOOP, '97), pages 220-242. Springer Verlag, 1997.

[10] KIENZLE, YANG YU, AND JIE XIONG. *On Composition and Reuse of Aspects.*

[11] PAWLAK, L.SEINTURIER, L.DUCHIEN, AND G.FLORIN. *JAC: A flexible solution for Aspect-Oriented Programming in Java.* In Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001), LNCS 2192, pp. 1-24, Springer, 2001.

[12] POPOVICI.A, T.GROSS, AND G.ALONSO. *Just-In-Time Aspects: Efficient Dynamic Weaving for Java.* AOSD 2003, Proceedings of the 2nd international conference on Aspect-oriented software development, Boston, Massachusetts.

[13] PAWLAK, L.DUCHIEN, G.FLORIN, AUBRY, L.SEINTURUER, AND MARTELLI. *JAC: An aspect-based distributed dynamic framework. Software Practise and Experience (SPE).*

[14] TARR.P, H.OSSHER, W.HARRISON, AND S.SUTTON. *N Degrees of Separation: Multi-dimensional Separation of Concerns.* In 1999 Intl. Conf. on Software Engineering, pages107-119, Los Angeles, CA, USA, 1999.

[15] WASIF GILANI, AND OLAF SPINCZYK. *A Family of Aspect Dynamic Weavers.* AOSD 2004 -Dynamic Aspects Workshop, Lanchester, UK,Mar.2004.

[16] WILLIAM HARRISON, AND HAROLD OSSHER. *Subject-Oriented Programming (a critique of pure objects).* In Proceedings of OOPSLA'93.

[17] XEROX CORPORATION. *The Aspectj Programming Guide.* online documentation, 2002. http://www.aspectj.org/.

[18] YOSHIKI SATO, SHIGERU CHIBA, AND MICHIAKI TATSUBORI. *A Selective, Just-In-Time Aspect Weaver.* Proceedings of the second international conference on Generative programming and component engineering, Erfurt, Germany, Year of Publication: 2003.